Using Domain-Specific Languages For Analytic Graph Databases

Martin Sevenich
Oracle Labs
martin.sevenich@oracle.com

Sungpack Hong
Oracle Labs
sungpack.hong@oracle.com

Oskar van Rest
Oracle Labs
oskar.van.rest@oracle.com

Zhe Wu
Oracle
alan.wu@oracle.com

Jayanta Banerjee
Oracle
jayanta.banerjee@oracle.com

Hassan Chafi
Oracle Labs
hassan.chafi@oracle.com

ABSTRACT

Recently graph has been drawing lots of attention both as a natural data model that captures fine-grained relationships between data entities and as a tool for powerful data analysis that considers such relationships. In this paper, we present a new graph database system that integrates a robust graph storage with an efficient graph analytics engine. Primarily, our system adopts two domain-specific languages (DSLs), one for describing graph analysis algorithms and the other for graph pattern matching queries. Compared to the API-based approaches in conventional graph processing systems, the DSL-based approach provides users with more flexible and intuitive ways of expressing algorithms and queries. Moreover, the DSL-based approach has significant performance benefits as well, (1) by skipping (remote) API invocation overhead and (2) by applying high-level optimization from the compiler.

1. INTRODUCTION

In recent years, the data management community as well as the data mining community, both in academia and industry, have been paying a lot of attention to the graph-based approaches in which graphs are used as fundamental representation for data modeling and data analysis. These approaches are very promising as modeling the data as a graph allows to capture arbitrary and dynamic relationships between data entities without the need of a predefined, rigid schema. Also by applying analysis algorithms on the graph data, non-immediate and non-obvious relationships can be discovered which could provide insights into the original data set.

Interestingly, two different types of graph processing systems have emerged with focuses on different aspects of graph processing. The first, *Graph Databases* [6, 9, 15] focus mainly on managing the dataset, modeled as a graph, in a persistent storage with some consistency guarantees. Graph databases usually define an API which allows users to conveniently access the graph data from the storage. However, it is the users who should come up with efficient implementations for their custom graph analysis algorithms out of such a low-level API. Furthermore, some studies observed performance issues in such implementations [34, 44].

The second type of systems are *Graph Analytics Engines*, frameworks that are specifically designed for fast execution of graph analysis algorithms [2, 29, 31, 33, 35]. When using such a framework, the user first needs to encode these graph algorithms as small computation kernels; the framework takes charge of running those kernels in parallel or distributed manners. Note that most of these engines load up the graph into the (distributed) memory before the analysis begins, due to performance reasons. Moreover, these engines assume that the original data remains static during the analysis, i.e. they do not consider consistency of the original data.

Both graph databases and analytic engines do not have a standardized way of implementing analytic algorithms. They expose an API to the users that they should use to implement their custom algorithms. This can be a non-intuitive task as the abstract description of the algorithm might be very different from the programming model the API uses. Also using an API can come with a significant runtime overhead as we will show in this paper. Furthermore the code is not portable as a different execution engine is likely to use a different API or even a different programming model.

In this paper, we present a new graph database system that is capable of both consistent management and efficient analysis of graph data.[1] We achieve this by tightly coupling a consistent *Graph Storage Engine* with an efficient in-memory *Graph Analytics Engine* (PGX – Parallel Graph analytiX). Our system provides two different domain-specific languages (DSLs) to deal with the shortcomings of an API based interface. We use Green-Marl [24] for graph analytic algorithms and PGQL (Property Graph Query Language) for pattern matching queries. Each language is specifically designed for its particular use case, therefore offering users an intuitive way of implementing their programs. As a result, this increases the overall usability of our system and the users productivity. The DSL programs are compiled inside our system, where the compiler can take advantage of the graph-specific semantics of the language and apply high-level optimizations. These optimizations - combined with the fact that the compiled DSL code avoids the overhead of an API - provides our approach with a superior performance compared to a purely API-based solution. Additionally the DSL programs can easily be ported to different platforms with no changes to the source code.

Proceedings of the VLDB Endowment, Vol. 9, No. 13

[1]The paper describes the design of a prototype implementation of the graph database system. Not all the features discussed here may be exposed in the final product.

Our specific contributions in this paper are as follows:

- We illustrate the basic architecture of our graph database system which is a tight integration of a graph analytics engine and a graph storage system. (Section 3)
- We adopt two domain-specific languages – one for imperative analytic algorithms, the other for declarative queries. We discuss how the users can make use of two languages synergistically in intuitive ways. (Section 4)
- We explain performance benefits from using DSLs. Particularly, we explicate various compiler optimizations that are enabled by high-level semantic information available from the DSL. We also show how this approach avoids the high performance overhead of an API-based implementation of analytic algorithms. (Section 5 & 6)
- We discuss the ongoing extension of our graph database system where we adopt different analytic engines (e.g. distributed processing). We also delineate how using DSLs allows a transparent migration. (Section 7)

2. BACKGROUND AND RELATED WORK

Graph Databases

There are two different types of commercial and open-source graph databases: First, there are graph databases [1, 11, 16] that adopt the classic *Resource Description Framework* (RDF) data model. The RDF model regularizes the graph representation as set of triples (or edges). In addition, when combined with semantic information, RDF is capable of applying powerful inference rules to the data. However, full-scale graph analyses, e.g. spectral analysis, have rarely been applied to RDF graphs. In the RDF model, even constant literals are encoded as graph vertices, and such artificial vertices can induce undesired noise to the outcome of the graph analysis.

Second, there are graph databases [6, 9, 15] that adopt the *Property Graph* (PG) data model in which vertices and edges in a graph can be associated with arbitrary *properties* as key-value pairs. Many of these PG databases tend to focus on providing a data access API while offering limited graph analysis support. Furthermore, graph algorithms implemented on top of such an API, may perform significantly worse than a direct in-memory implementation [34, 44].

Our current system adopts the PG data model, providing efficient in-memory analyses on it. However, we are also investigating ways to apply such graph analyses on the RDF data model.

Domain-Specific Languages for Graphs

There is no standardization for graph languages yet and so there are multiple graph specific DSLs.

Our system adopts two DSLs – Green-Marl [24] and PGQL – as front-ends with which users describe their custom graph algorithms or pattern-matching queries. Note that a few other DSLs have been proposed for graph processing.

There are other languages designed for querying patterns in the graph: For example Cypher [5] for the PG model and SPARQL [12] for the RDF model. These languages allow the user to specify a graph pattern of interest; the graph database then finds matching subgraphs or matching graph elements. However, these languages are not suitable for graph analysis algorithms as those algorithms are more than a simple graph pattern – they are complex procedures.

Gremlin [14] is a language designed for graph traversal. Although it is possible to do so, Gremlin has certain issues when it comes to implement graph analysis algorithms and we think that Green-Marl is more intuitive for describing graph algorithms, due to its imperative programming style and the ability to apply graph specific optimizations such as the ones presented in this paper.

Graph Libraries and Graph Processing Frameworks

There are some systems that focus on the fast execution of graph analysis algorithms. First, there exist many different graph libraries for different programming languages. Examples include NetworkX [10] (Python), Jung [8] (Java), SNAP [13], Stinger [20] and the Boost Graph Library [4] (C++). Essentially, these libraries provide their own graph data structures; for custom graph algorithms the users have to create their own parallel implementation on top of the library's data structures.

Second, there are graph processing frameworks which consider parallel or distributed execution of graph algorithms [2, 29, 31, 33, 35]. Galois [35] allows fast propagation of vertex-oriented events in parallel, shared-memory environments, while GraphLab [31] implements a similar concept in distributed environments. Giraph [2], Pregel [33] and Pegasus [29] are scalable graph processing systems based on a Map-Reduce environment.

Note that each of the above frameworks requires a special, restricted programming model, to perform parallel or distributed execution. Therefore, users have to redesign their graph algorithms accordingly, which can be a daunting task for complicated algorithms. Our approach, however, allows the user to write an intuitive, high-level DSL program and leaves it to the compiler to parallelize it.

More importantly, neither these graph libraries nor the graph processing frameworks concern consistent data management at all. In this paper, we show that by tightly coupling a graph database with a graph analytic engine, one can get the best of both systems.

TinkerPop [3] is an open-source graph computing framework, providing different interfaces and frameworks for graph analytic systems and graph databases. This includes Blueprints, an API for the property graph model, Pipes, a data flow framework, the Gremlin graph traversal language, the Frames object-to-graph mapper, the graph algorithms package Furnace and Rexter - a graph server. Many graph databases such as Neo4j [9] and Titan [15] comply to APIs defined by TinkerPop which allows using its capabilities on top of the database. The DSL based approach in our system offers a more intuitive way to implement graph analytic algorithms compared to such an API since it is not bound to the Pregel computation model. Also it provides better performance as it allows high-level compiler optimizations while avoiding the overhead of repetitive calls to a system API.

Graph Processing in Relational Databases

Several systems advocate the use of relational database systems (RDBMS) instead of a specialized graph processing system to benefit from the fact that RDBMS are already widely used for analytic purposes and are well understood and optimized. Grail [21] for example stores the vertex and edge data in tables in a relational database while offering a vertex-centric API to the user for writing queries. These are translated into SQL and then executed by the RDBMS.

SQLGraph [40] also uses a relational database to store the graph data but it only stores the graph topology in the relational storage. The vertex and edge data is kept separately in a JSON storage. The system uses Gremlin [14] as a language to let users describe graph queries that are transformed into SQL before being executed by the system. Grail and SQLGraph rely on the RDBMS alone to optimize the generated SQL statements, but due to the nature of SQL this

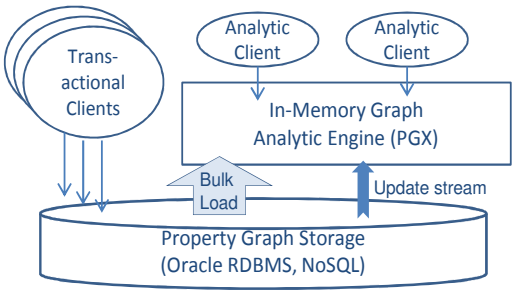

Figure 1: The overall design of our system

does not include optimizations that use graph-specific knowledge. The compiler in PGX on the other hand uses the graph-specific knowledge of the DSL semantics to optimize the generated code.

3. GRAPH DATABASE SYSTEM: THE ARCHITECTURE

3.1 System Overview

In a nutshell, our graph database system is a tight integration of a graph storage and a graph analytics engine. The graph storage enables robust management of large graph data sets under highly concurrent transactional workloads, while the graph analytics engine enables fast parallel execution of analytic algorithms on the graph data. Figure 1 illustrates the overall architecture of our graph database system.

Our graph storage provides the Property Graph [3] data model on top of industry-proven data management systems, including Oracle RDBMS, Oracle NoSQL and Apache HBase. This approach saves us from *re-inventing the wheel*, or we do not need to build whole new database engine specialized for the graph data model; complex enterprise requirements such as security, access control, transactions and scalability have already been addressed in these RDBMSs [44].

PGX, our graph analytic engine focuses on in-memory parallel execution of graph analytics algorithms, exploiting the large memory capacity and multiple CPU cores in modern server systems. Even though the graph data is maintained in tabular forms in the storage, PGX utilizes a more specialized data structure for its in-memory representation that keeps explicit edge lists.

Note that graph analysis algorithms typically perform a lot of neighborhood iterations and traversals, which naturally induces numerous non-sequential data accesses. Therefore, running graph algorithms directly on top of an in-memory graph data structure provides significant performance benefits over other approaches like repeated computation of joins on a tabular representation [21], frequent (random) accesses to block storage [37], and exchanging many short messages with remote machines [31]. Section 7 discusses on-going extensions of our system for handling very large graph instances that do not fit in a single memory.

Overall, by combining a robust graph storage and a fast graph analytics engine, our graph database system provides the users with the best of both worlds: maintaining large data sets using an intuitive graph data model and applying graph analytics on them in an efficient manner.

3.2 Features and Interfaces

Graph Loading and Delta Update

In order to apply analytic algorithms on the graph data, the data in the storage first needs to be loaded into the analytic engine, as illustrated in Figure 1. Our graph database system provides a fast bulk loading mechanism for this large data movement – even graphs with billions of edges can be loaded in tens of minutes.

In addition, our graph database system provides an optional feature of *delta updates*. That is, once the large graph data is loaded into the analytic engine (PGX), subsequent changes to the data in the storage are tracked by the analytic engine. Either by the user's explicit requests or by predetermined periodic events, a new graph instance is created by applying these changes to the previously loaded graph instance.

Finally, we provide various options for loading only the relevant subset of the large data into memory. For instance, the user can provide a list of properties that are to be loaded into memory, allowing to keep other properties in the storage as the analysis would not require them. The user can also specify a short filter expression for defining a subgraph to be loaded into memory. For example, the following edge-filter expression defines a subgraph where all edges contain at least one RED vertex on either side:

```
src.color = 'RED' or dst.color = 'RED'
```

Concurrent Clients, Snapshot Consistency and Mutation

Our graph database system supports multiple concurrent clients in a scalable manner. First of all, transactional clients are separated from analytic clients and served by different engines (see Figure 1). Therefore, a small number of long-running computation-heavy analytic workloads are handled very differently than a large number of short and frequent IO-intensive transactional workloads.

Moreover, the PGX graph analytic engine provides an isolated, consistent snapshot view of the graph data to each analytic client. That means, each client can work on a graph instance that is consistent to a certain version of the data in the storage. For the sake of analysis, the client can define its own additional properties and modify their values. The client can even mutate the graph itself, i.e. add or remove vertices and edges. All these changes, however, are not visible to other concurrent clients that are working on the same graph instance. To make changes permanent, the client has to commit them to the graph storage; the changes will be only visible to other clients when they choose to fetch a more recent graph instance from the storage, potentially through the delta update mechanism.

When there are multiple concurrent analytic clients, it is possible to deploy multiple PGX analytic engines with a single graph storage. Again, our design guarantees that each PGX instance holds a consistent snapshot of the graph data, as the data storage becomes the reliable *source of truth*.

Execution Modes and Interface Layers

PGX provides two different modes of execution for analytic clients: remote execution mode and embedded execution mode. In remote mode, the PGX analytic engine is wrapped in a web container and deployed as a web service. Each remote client can make a connection and submit remote requests to the server. Typically, the PGX engine resides in a server-class machine exploiting its large memory capacity and high computation power. In embedded mode, on the other hand, the client application runs in the same process space as PGX. Essentially, PGX becomes a graph library in this mode.

PGX makes it trivial to switch between remote execution and embedded execution through its careful API design. While both a remote client and a local client would invoke exactly the same

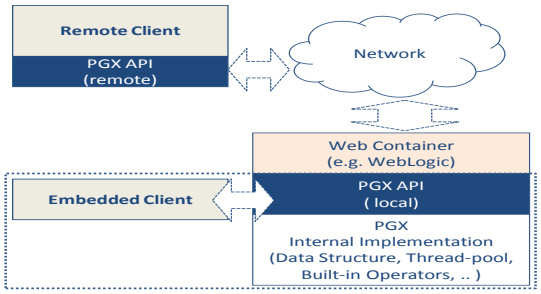

Figure 2: Remote and local execution modes and PGX API layers

interface methods, the remote ones are transparently mapped into server-side REST API invocation. As a matter of fact, all the methods in the PGX API support asynchronous execution (e.g. using `Future` in Java7) in addition to their synchronous counterparts. Consequently, a local PGX client can seamlessly run remote as well, just by linking it to a different API implementation. Figure 2 illustrates the execution modes and the API layers.

The PGX API itself consists of two layers. The first one is the high-level API package which provides a set of fixed functionalities including built-in analysis algorithms, graph mutation operators, and graph loading/exporting mechanisms. In other words, the methods in this package initiate big computations in the server side. The second one is the low-level API package which provides fine-grained control over each `vertex` and `edge` of the graph. Basically, this API is designed as convenient methods for small and short interactions between client and server. For instance, after identifying the top 10 Pagerank vertices, the user may look up additional properties of them.

The challenge is, however, when the user wants to execute a custom graph algorithm which is not pre-built in the package. Although users can always implement their custom algorithm using the low-level API (i.e. `vertex` and `edge`), this approach may introduce a significant performance problem because the (remote) API overhead gets accumulated over each vertex and edge access. Note that even the embedded clients could suffer from this performance overhead, because PGX internal graph representation is indeed different from the convenient object-oriented one in the low-level API.

Instead, we adopt a Domain-Specific Language (DSL)-based approach in PGX. More specifically, users can write up their graph algorithms in a DSL and submit them to the server, while the server compiles and executes them efficiently without any API overhead. Note that this approach certainly follows an important design principle in modern computing systems: *move computation instead of data*. Section 4 explains our DSLs while Section 6 shows the performance improvements from using DSLs instead of an API. As a final note, all the built-in algorithms in our release packages are in fact compiled from DSL code.

4. DOMAIN-SPECIFIC LANGUAGES

In this section we discuss the two domain specific languages that we adopted – Green-Marl for graph analysis and PGQL for graph queries and pattern matching – and how both languages can be used together.

4.1 Green-Marl

Green-Marl [24] is a domain specific language designed specifically to express graph analysis algorithms. The language supports

```
1  procedure pagerank(G: graph, e,d: double, max: int;
2                     pg_rank: nodeProp<double>) {
3    double diff;
4    int iter = 0;
5    double N = G.numNodes();
6    G.pg_rank = 1 / N;
7    do {
8      diff = 0.0;
9      foreach (t: G.nodes) {
10       double val = (1 - d) / N + d *
11         sum(w: t.inNbrs) {w.pg_rank / w.degree()};
12       diff += | val - t.pg_rank |;
13       t.pg_rank <= val;
14     }
15     iter++;
16   } while (diff > e && iter < max);
17 }
```

Figure 3: Pagerank algorithm implemented in Green-Marl [24]

```
1  SELECT friend.name, friend.age
2  FROM friendshipGraph
3  WHERE
4    (m WITH name = 'Mario') -[:likes]->(friend),
5    (l WITH name = 'Luigi') -[[:likes]*1..2]->(friend),
6    friend.age >= m.age + 2
7  ORDER BY friend.name
```

Figure 4: Example PGQL query, returning the friends of Mario that are also friends, or friends of friends, of Luigi. Friends that are at least two years older than Mario are filtered out.

graph-specific data entities as intrinsic data types: graph, vertex, edge as well as vertex property and edge property. It also provides languages constructs for different graph traversals and iterations, such as Breadth-First Search (BFS), Depth-First Search (DFS), incoming neighbor iteration, outgoing neighbor iteration, etc.

Like most mainstream languages, Green-Marl is an imperative language that assumes a global shared memory. Combined with the high-level graph-specific language constructs, this allows users to implement a graph analysis algorithm in a straightforward manner, since the Green-Marl code resembles the algorithm description; the users need not rewrite their algorithms with certain artificial constraints such as in vertex-centric programming models. As an example, Figure 3 shows the Green-Marl implementation of the Pagerank algorithm.

The high-level language constructs in Green-Marl expose the inherent parallelism to the compiler which is then able to generate a highly parallel executable. Moreover, the DSL compiler can apply very specialized optimizations that a general purpose compiler cannot do, because the DSL compiler understands the specific semantics of the language constructs. We present some of these optimizations in Section 5.

Note that the compiler applies these optimizations automatically; the user only focuses on the high-level implementation of algorithms while the compiler provides the performance via parallelization and optimization. Furthermore, whenever new optimizations are added to the compiler, existing algorithms get performance benefits automatically, if the new optimizations are applicable to them. Section 5 discusses how the same optimizations are applied to different algorithms.

4.2 PGQL

PGQL (Property Graph Query Language) is our proposal of a pattern-matching query language tailored for Property Graphs. A graph pattern-matching query is a query to find all instances in the

given data graph that match to the specified graph pattern. The detailed syntax and semantics of PGQL are outside the scope of this paper as we are preparing a separate publication about it. Instead, we give a short introduction to PGQL here.

There is the need for a new query language as there does not exist a theoretically sound, yet feature-complete declarative query language for the PG data model at the moment. Cypher for example is missing fundamental graph querying functionality. It supports subgraph *isomorphic* queries, but not the more general class of the subgraph *homomorphic* queries. Also it does not allow for the well-studied class of the regular path query (RPQ). Finally, although Cypher allows for updating of graphs, it does not support constructing new graphs, which is essential for graph transformation applications and typical database functionality such as SQL-like *Views*. PGQL overcomes these limitations.

The syntax structure of PGQL resembles that of SQL. Basically, a PGQL query is composed of three clauses (`SELECT`, `FROM`, `WHERE`) followed by optional solution modifier clauses such as `ORDER BY`, `GROUP BY`, and `LIMIT`. The `FROM` clause can be omitted when there is only one graph instance. Additionally, PGQL includes special operators for graph pattern matching: vertex matching, edge matching, path matching, etc. These matching operators are placed in the `WHERE` clause along with other expressions to construct predicates.

Figure 4 shows an example PGQL query which finds patterns from a data graph named `friendshipGraph` (line 2). In the `WHERE` clause, the query matches a vertex `m` that has a property `name` with value `'Mario'`. The vertex `m` has an edge whose label is `'likes'`. The destination vertex of this edge is referred as vertex `friend` (line 4). Similarly, the query matches another vertex `l` with its `name` being `'Luigi'`. The vertex `l` is also connected to the vertex `friend` but through a path; the path is composed only of edges whose label is `'likes'` and the (hop-)length of the path is between `1` and `2` inclusively (line 5). Line 6 dictates that the value of property `age` in vertex `friend` is larger than or equal to that of vertex `m` by two. All the instances that match with this pattern are first sorted by `name` values of `friend` vertices (line 7), before the `name` and `age` property values of `friend` vertex are returned (line 1).

Note that the above example query contains a path matching operator. In general, PGQL supports regular path queries (RPQs), while the path matching operator can be used differently for finding reachability vertices or for enumerating all (shortest) paths. More specifically, PGQL supports the class of extended conjunctive RPQs (ECRPQs) [18] but with arbitrary expressions over edges (not just over edge labels) and optional restrictions on path lengths.

Just like in SQL, the result of a PGQL query forms a tabular "result set" with variables and their bindings. However, PGQL also has intrinsic data types for graph-specific entities like *Vertex*, *Edge*, *Path* and *Graph* – the binding can be any of these graph-specific types. In addition, PGX also provides aggregate methods that merge matches in the result set into another graph instance.

4.3 Combined Usage of Two Languages

As discussed so far, PGX adopts not one but two DSLs. This is to give users the most intuitive programming model for each use case: imperative for analysis and declarative for queries. Imperative languages provide constructs for fine-grained control flows and (intermediate) value computation and management, which is essential for writing algorithmic procedures. Declarative languages on the other hand, make it very easy to specify a pattern to be matched on, even complicated ones; the execution engine can make intelligent choices to map the query into a sequence of predefined operators. Consider the analysis example in Figure 3 and the query example in Figure 4. Even with these simple examples, it is not obvious how to re-write the analysis in Figure 3 with a query language like PGQL, and vice versa.

However, despite their many differences, both languages share a lot of common parts in the runtime implementation. Both languages work on the same graph data representation at least. All PGX system resources (thread-pools, scheduler, memory management) are naturally shared as well. More importantly, two runtime systems share a single implementation for many performance-critical operations: breadth-first traversal, common-neighbor iteration, top-k value finding, etc.

Furthermore, the combined usage of two languages can provide the users with more benefits than each individual language. Users can first use PGQL to extract a sub-graph from the original data and then run a Green-Marl analysis on the result. For instance, the who-to-follow analysis from Twitter [23] can be performed in following manner:

1. Given a vertex v_0, do Personalize Pagerank (with a Green-Marl program) starting from the vertex.
2. Use PGQL to identify the top *T* closest vertices to v_0.
3. Create a bipartite sub-graph using those *T* vertices as left-hand-side (LHS) vertices and their neighbors as right-hand-side (RHS) vertices. PGX provides an API for this.
4. Run the SALSA algorithm (another Green-Marl program) to compute relative importance scores on each type of vertex.
5. Use PGQL to identify the top *K1* LHS vertices and the top *K2* RHS vertices.
6. The RHS vertices become a recommendation list.

Of course this also works the other way – a PGQL query can include the values computed by a Green-Marl program in the search. As an example, consider the following scenario which tries to find low-centrality vertices that bridge two high-centrality vertices:

1. Run Pagerank on the graph and compute the 5 percentile and 95 percentile Pagerank values t_5 and t_{95} (Green-Marl programs).
2. Use PGQL to find each vertex *v* such that the Pagerank value of *v* is less than t_5. However, *v* is connected to *w* and *u* both of which have Pagerank values larger than t_{95}. Also there is no edge between *w* and *u*.

5. COMPILER OPTIMIZATIONS

The Green-Marl compiler uses program analysis and knowledge specific to the graph domain to perform a variety of optimizations on given Green-Marl code. In this section we present and describe a number of these optimizations. Section 5.1 and 5.2 are based on previous work, while the following sections focus on new contributions. Table 1 contains a variety of algorithms and the optimizations presented in this paper that can be applied to the code. Note that often applying a single optimization has only a small or even negative impact on the performance, but combining multiple optimizations can lead to significant performance improvements. See Section 6.3 for an evaluation on their impact on the performance.

5.1 Basic Graph Optimizations

The Green-Marl compiler applies several graph specific optimizations presented in previous work [24]. This includes system and architecture independent optimizations (e.g. loop merging) and optimizations that are specific to the architecture the program is executed on (e.g. selection of parallel regions).

5.2 Common Neighbor Iteration

In our previous work [38] we presented an optimization where the compiler identifies iterations over common neighbors of two

Algorithm	MB	DS	PM	DP	CI
Adamic Adar [17]					✓
Betweenness-Centrality [32]	✓				
Closeness-Centrality [22]	✓				
Dijkstra [19]		✓			
Fattest-Path [28]		✓			
Kosaraju [19]		✓			
PageRank [36]			✓	✓	
Soman and Narang [39]			✓	✓	
Tarjan [42]		✓	✓		
Triangle Counting [41]					✓

Table 1: Graph analysis algorithms and applicable optimizations. MB: Multi-Source-BFS, DS: Data Structure Specialization, PM: Property Merging, DP: Degree Precomputation (including Inverse-Degree Precomputation), CI: Common Neighbor Iteration.

vertices and uses a specialized algorithm to perform this iteration. Depending on the results of a program analysis, it also generates code to prune the search space for these common neighbors which significantly reduces the work to be done. The combination of these two optimization steps significantly improves the performance of certain algorithms like Triangle Counting.

5.3 Multi-Source-BFS Transformation

Multi-Source Breadth-First Search (MS-BFS) is a technique developed by Then et al. [43] to perform multiple BFS traversals at the same time, starting from several different roots. These BFSs will be packed together in a so called batch and share common parts of the traversal, therefore reducing the overall amount of random memory accesses while making better use of the memory prefetcher and SIMD instructions. It has been shown, that MS-BFS can greatly increase the performance, especially on small-world graphs as they can be often found in social networks.

The downside of using MS-BFS is an increased code complexity since the program has to make sure that each BFS iteration in a batch behaves as if it ran independently. Also MS-BFS can significantly increase the memory consumption of the program - namely by a factor up to the size of one batch - since copies of certain variables have to be created which can be very costly for properties. This extra memory consumption can easily exceed the available system memory, even for mid-sized graphs. Dealing with these issues can be a challenging and error prone task.

In the following we describe how the Green-Marl compiler can address these problems in an automatic and transparent way.

MS-BFS Auto-Transformation

The Green-Marl compiler can automatically transform code to use MS-BFS. That means the user can write the code using the normal BFS iteration that is built into Green-Marl, and the compiler identifies whether MS-BFS can be used and if this is the case it takes care of all the code adjustments needed to use MS-BFS correctly. This gives the user the benefits of MS-BFS without the burden of having to deal with the implications it has to the code.

Consider the example code below:

```
foreach (s: G.nodes) {
  long bfsSum = 0;
  nodeProp<double> prop;

  foreach(n: G.nodes) {
    n.prop = 0;
  }
  inBFS(v: G.nodes from s) {
    v.prop++;
    bfsSum = bfsSum + 1;
  }
}
```

The compiler searches for a pattern with a `foreach` loop where the iterator is used as source vertex for a BFS iteration. The iteration can be over all vertices in the graph or just a subset. This code is then transformed to use MS-BFS where each BFS in the batch is identified using an index variable. Note that the syntax below is not part of the Green-Marl language and is just used for illustration purposes.

```
ms_bfs_foreach (s: G.nodes) {
  ...
  in_ms_bfs(v: G.nodes from s | batchIndex) {
    ...
  }
}
```

The next step is to identify all scalar variables that are visible to each BFS and therefore have to be protected - e.g. `bfsSum` from the example. For each BFS in the batch a copy of these variables is created and stored in an array. Outside of the BFS, writes to these variables are turned into a loop where the value for each BFS is written. Inside the BFS, the batch index is used for reads and writes. Property variables are treated slightly differently. Instead of creating a copy for each BFS, the size of the property will be multiplied by the batch-size to ensure that the data lies packed in memory.

```
ms_bfs_foreach (s: G.nodes) {
  long bfsSum[batchSize];
  nodeProp<double * batchSize> prop;

  for (0 ≤ batchIndex < batchSize) {
    bfsSum[batchIndex] = 0;
  }

  foreach(n: G.nodes) {
    for (0 ≤ batchIndex < batchSize) {
      n[batchIndex].prop = 0;
    }
  }

  in_ms_bfs(v: G.nodes from s | batchIndex) {
    v[batchIndex].prop++;
    bfsSum[batchIndex] = bfsSum[batchIndex] + 1;
  }
}
```

Auto-Memory Tuning

Memory consumption can become a problem when using MS-BFS as multiple MS-BFS instances might run concurrently and a copy of each property might be necessary per thread and per item in the batch. Depending on the graph size, this can easily exceed the available memory of a machine, even for mid-sized graphs.

There are two ways to deal with this issue that can be applied automatically by the Green-Marl compiler: reducing the number of threads or reducing the batch-size. In the following we will discuss how to reduce the batch-size, but reducing the number of threads works similarly. Note that each approach has its own advantages and disadvantages and decisions have to be made based on the target system and its requirements. For example reducing the number of threads while using an optimal batch-size can be advantageous if the idle threads can be used for other tasks while reducing the batch-size might be beneficial to utilize all system resources.

The compiler analyzes the code to determine copies for which properties have to be created. It then adds code that - at runtime - computes the maximum batch-size for which the required memory will still fit in the memory currently available to the system. The memory required by all vertex properties is the sum of the products of the type-size, the number of vertices, the number of threads and the batch-size. The memory for all edge properties can be computed accordingly. The batch-size will be set to the minimum of the computed batch-size and the size of the SIMD registers of the machine.

5.4 Data Structure Specialization

Traditional object oriented languages provide abstract data structures such as `list` or `set`, but they leave it to the programmer to select the actual implementation for example a `linked-list` or a `hash-set`. This choice can be crucial for the program performance as different implementations perform better or worse in different use cases. Green-Marl provides abstract data structures as well, but the compiler chooses the implementation using code analysis and different optimization techniques that also allow the compiler to rewrite parts of the code to leverage the benefits of the particular implementation. In the following we describe two such optimizations that are based on selecting specialized data structure implementations.

Priority Map Selection

Green-Marl provides a `map` type that uses a `hash-map` as default implementation. The Green-Marl `map` offers the `getKeyWithSmallestValue()` and `getKeyWithLargestValue()` functions to retrieve the key associated with the smallest or largest value in the map. In the following we will describe the optimization for keys associated with the smallest value, but it works identically for the largest value.

We implemented a different, heap-based, `priority-map` that has a $\mathcal{O}(1)$ runtime guarantee for `getKeyWithSmallestValue()` and `getKeyWithLargestValue()`. It also provides a function `removeSmallest()` and `removeLargest()` for deleting the smallest or largest value; running in $\mathcal{O}(\log n)$. Note this function is not available to the user as it is not part of the language itself, but the compiler uses it for optimization purposes.

To determine whether to use a `priority-map` implementation, the compiler looks at all variables that are used as keys to read or remove values. If all these variables are being assigned by using the `getKeyWithSmallestValue()` or `getKeyWithLargestValue()` function - i.e. we only read or remove the smallest value in the map - then we choose the `priority-map` implementation. Additionally the code can be rewritten to call `removeSmallest()` or `removeLargest()` directly instead of `remove(key)` which often allows to optimize away the call to `getKeyWithSmallestValue()` altogether. The following Green-Marl code example illustrates that:

```
map<node, double> m;
...
node n = m.getKeyWithSmallestValue();
m.remove(n);
```

becomes

```
map<node, double> m;
...
m.removeSmallest();
```

This optimization can significantly improve the performance and scalability (see Section 6.3) as it reduces the worst-case runtime of operations like `getKeyWithSmallestValue()` from $\mathcal{O}(n)$ to a constant factor.

Stack Selection

`nodeSeq` and `edgeSeq` in Green-Marl allow to store vertices or edges while preserving the insertion order. Their default implementation is based on a `linked-list` as it provides good performance for most use cases. In some cases the compiler can choose an `array-list` based implementation instead which - in practice - provides better performance for read operations and writes to the end of the list due to the data being stored aligned in memory. Also `array-lists` tends to generate fewer objects compared to a `linked-list`.

The compiler finds sequence variables where all modifications only occur to the back via `pushBack()` or `popBack()`, i.e. the sequence is used as a stack. Note that sequences do not support modification to locations other than the front or back. For example the compiler will choose an `array-list` for the sequence in the following snippet.

```
nodeSeq seq;
for (n: G.nodes) {
  seq.pushBack(n);
}
```

Additionally the compiler finds sequences where only the front is modified using `pushFront()` or `popFront()`. It then rewrites the code to modify the back of the sequence instead (e.g. replace `pushFront()` with `pushBack()`) and changes reads at the front to reads at the back and vice versa. Also the direction of all iterations over the sequence are inverted.

```
nodeSeq seq;
for (n: G.nodes) {
  seq.pushFront(n);
  node m = seq.back();
}
for (n: seq.items) {    // forward iteration
  ...
}
```

becomes

```
nodeSeq seq;
for (n: G.nodes) {
  seq.pushBack(n);
  node m = seq.front();
}
for (n: seq^.items) {    // reverse iteration
  ...
}
```

This transformation then allows applying the same optimization described above without changing the semantics of the program.

5.5 Property Merging

PGX implements vertex (edge) properties column wise as an array of length $\mathcal{O}(N)$ ($\mathcal{O}(E)$) which provides excellent performance when iterating over the values in a single property. Because arrays of different properties are stored in different memory locations, accessing multiple properties at once - e.g. in the same expression - can have a negative impact on the performance due to the two reads/writes required in different locations in memory due to an increased number of cache misses. In such a case, the compiler can transform the code from a column-oriented property representation to a row-based representation, i.e. the values of multiple properties will be stored consecutively in memory for one vertex (edge).

Since this optimization adds runtime overhead, the compiler has to use a heuristic to determine whether and to which properties it will be applied. Note that this optimization is performed twice, once for vertex properties and a second time for edge properties as they cannot be merged together due to the different dimensions.

Merge Heuristic

First the compiler identifies all sets of properties that are used together - i.e. in the same expression including sub-expressions. These sets represent the candidates for being potentially merged into a single property. One property can appear in multiple sets, but usually the number of sets is small since typical graph algorithms use only a small number of properties.

These candidate sets are then ranked according to a cost value λ that is calculated by the compiler. In the following we list the factors that influence the cost value. How each of these factors influence λ is platform and system specific.

- A larger set size will increase λ since the benefits from merging are smaller while increasing the overhead.
- Properties in the set that are input- or output-arguments of the procedure will increase λ because they require extra code that copies the values from the input into the merged property and the values from the merged property into the output. Also they consume extra memory while local properties can be replaced entirely by the merged property.
- Sets that would require the merged property to include data padding increase λ due to the additional memory consumption.
- Each occurrence of an expression that involves all properties of a candidate set will decrease λ by a certain value. This value is multiplied by a factor depending on whether the expression is nested inside of one or multiple loops such as `foreach` or `while` loops. The more expressions involving the set, the more potential benefit is promised by merging it.

The compiler will then perform the following until there are no candidate sets left or the top-ranked candidate set has a positive λ value: Remove the top-ranked candidate set and merge its properties; then remove all other candidate sets that contain at least one property from this set. This step is necessary because one property cannot be part of multiple merged properties.

Code Transformation:

When merging a candidate set, the compiler will first create a new type with a member for each property in the set. Consider the following Green-Marl code snippet:

```
nodeProperty<double> p1;
nodeProperty<int> p2;
foreach (n: G.nodes) {
  n.p1 = 2.0 * n.p2;
}
```

Assuming `p1` and `p2` are selected for merging, then the following type would be created.

```
type p1_p2_type {
  double p1;
  int p2;
}
```

The compiler then rewrites the original code, creating a new property of the merged type, replacing the original properties and also rewriting all reads and writes to the original property.

```
nodeProperty<p1_p2_type> merged;
foreach (n: G.nodes) {
  n.merged.p1 = 2.0 * n.merged.p2;
}
```

Note that if the original properties are an input or output of the procedure, then they cannot be replaced. Instead code will be generated that for inputs copies the values from the input into the merged property at the beginning of the procedure and for outputs copies the values from the merged property to the output before the procedure returns. See the following code snippets for an example:

```
procedure foo(G: graph, in: nodeProperty<double>
                      ; out: nodeProperty<int>) {
  ...
}
```

Assuming `in` and `out` are merged, then the code becomes

```
type in_out_type {
  double in;
  int out;
}
```

```
procedure foo(G: graph, in: nodeProperty<double>
                      ; out: nodeProperty<int>) {
  nodeProperty<in_out_type> merged;
  foreach (n: G.nodes) {
    n.merged.in = n.in;  // copy from input
  }
  ...
  foreach (n: G.nodes) {
    n.out = n.merged.out;  // copy into output
  }
}
```

Precomputation of Vertex Degree

Computing the degree (i.e. the number of neighbors) of a vertex requires two memory lookups in the CSR structure we use for graphs and subtracting the values. The Green-Marl compiler can generate code to precompute these values for every vertex and store the results in a new property. Since this introduces overhead at runtime for creating and initializing the property, the compiler will only perform this code transformation if the resulting property would be subject to merging. This benefits the performance - and thus offsets the extra overhead - because it is only applied if the degree and other properties are frequently used together in the same expressions. So instead of having to access different memory locations for the degree and the properties, the values are stored adjacent in memory.

The transformation works as follows: First a new vertex property of type `int` is generated and initialized with the degree of each vertex. Then all degree lookups in the code are replaced by a lookup in the new property. Note that we only do this if the original expression was using floating point semantics for the division to prevent altering the semantics of the code. See the following Green-Marl code snippet for an example:

```
...
foreach (n: G.nodes) {
  int i  = 5 + n.degree();
}
...
```

becomes

```
nodeProperty<int> degree;
foreach (n: G.nodes) {
  n.degree = n.degree();
}
...
foreach (n: G.nodes) {
  int i = 5 + n.degree;
}
...
```

Inverting the Degree

Under certain conditions the above transformation is applied differently, namely if the degree-lookup only appears in expressions of the form `X / node.degree()`. In that case the new property is generated to be of type `double` and the values are initialized with the inverse of the degree.

```
nodeProperty<double> inv_degree;
foreach (n: G.nodes) {
  n.inv_degree = 1.0 / n.degree();
}
```

Instead of just replacing the degree lookups with lookups in the property, the original expression will be transformed from a division into a multiplication. For example consider the following code snippet

```
foreach (n: G.nodes) {
  double d = 1.0 + e / n.degree();
}
```

becomes

```
foreach (n: G.nodes) {
  double d = 1.0 + e * n.inv_degree;
}
```

This improves the performance by reducing the number of floating point division operations as they are more computational expensive than floating point multiplications [45]. But more importantly it can reduce contention in the system and therefore make better use of the systems threads. Many modern multicore CPUs have one floating point unit per core - but not all of them are capable of performing divisions [7] so multiple threads share the ones that are able to perform divisions. This can lead to congestion if the program performs many floating point divisions concurrently.

6. EVALUATION

In this section we present the performance experiments we conducted using different algorithms and real-world graph data sets. First we will discuss our methodology before we compare algorithms implemented using our API versus algorithms implemented and compiled in the Green-Marl DSL. Finally we show the performance impact of the compiler optimization presented in Section 5.

6.1 Methodology

We ran all our experiments on Intel Xeon E5-2699 (Haswell) machines with 36 2.30 GHz cores on 2 sockets. The machines contained 384GB of RAM and were running 64bit SMP Linux 2.6.32. The Java programs were compiled using Java version 1.7 and executed on the 64bit HotSpot Server VM.

We ran the code once to trigger the JIT optimizations before we measured the time using the average of 5 iterations.

Table 2 contains the real-world graph data sets used in the experiments including a short description, its origin and some characteristics.

6.2 API vs Compilation

In this section we show the performance overhead introduced by using different layers of API and compare it with the performance of compiled Green-Marl code. For this purpose we implemented the well known Pagerank algorithm in multiple ways to use different layers of the API to show the overhead introduced by them. Note that an operation like accessing a vertex becomes a request within the underlying API layer. This is necessary for PGX to support the features we described in Section 3.2. Figure 5 gives a brief overview of the different API layers in PGX. The Green-Marl code we used for Pagerank in these experiments is the same as in Figure 3.

- **Frontend:** This implementation of Pagerank uses the standard API and runs on top of the Frontend layer of PGX.
- **API:** This is basically the same implementation as Frontend, except that all API layers that deal with scheduling and consistency were stripped off.
- **Green-Marl:** This implementation of the Pagerank algorithm uses the Green-Marl DSL and submits the program to PGX where it gets compiled and executed inside the system. The compiled code can bypass all API layers and access the in-memory data directly.
- **Remote:** The Remote implementation uses the exact same code as the one using the Frontend layer, except that it is running in a server-client setting.

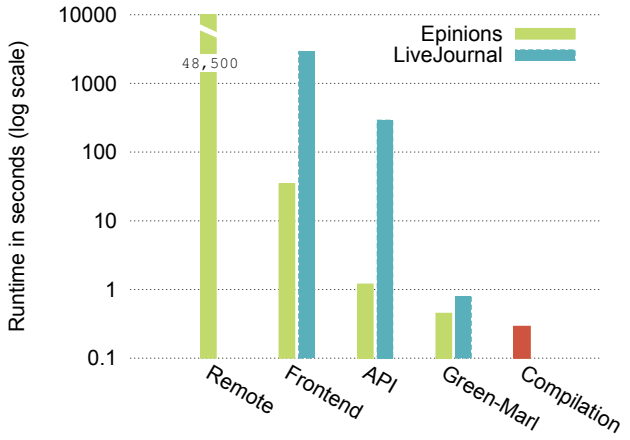

Figure 6: Absolute runtime of different Pagerank implementations on Epinions and LiveJournal graph data. The time for Green-Marl includes the compilation time. All numbers are from single-threaded execution.

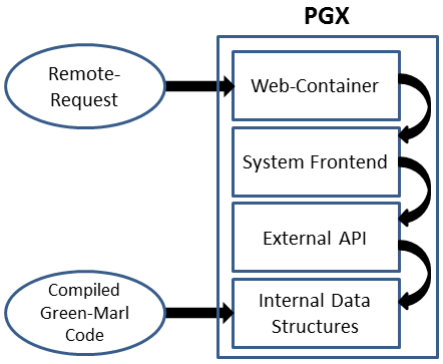

Figure 5: Overview of PGX API Structure

All experiments were executed single-threaded to avoid having different scaling behaviors interfere with the results. Figure 6 shows the absolute runtime of the different implementations on the Epinions and the LiveJournal graph. The Green-Marl numbers include the compilation time from the Green-Marl and Java compiler; all other numbers are runtime only. For reference we show the compilation time of the Green-Marl code separately in Figure 6. To avoid any network latency, we ran the client and server instance on the same machine for the remote implementation. We ran this version on the Epinions graph for a few iterations only and extrapolated the total runtime - also we only used the Epinions graph for this version.

As can be seen, the Green-Marl program outperforms all other implementations - for larger graph instances like LiveJournal by several orders of magnitude. It does so despite the fact that compiling the program makes up a significant portion of the runtime; for the smaller Epinions graph it even takes longer than the actual execution of the program. This clearly shows that the one-time cost of compiling the DSL program outweighs the accumulated runtime cost introduced by repeated API calls as they happen in the other implementations. The results also show how each extra layer of API introduces a certain overhead which basically slows down the program by at least one order of magnitude per layer - even more when adding the remote overhead.

Graph	Number of Vertices	Number of Edges	Description	Source
Epinions	75,879	508,837	Who-trusts-whom network of epinions.com	[13]
LiveJournal	4,848,571	68,993,773	Social relations in an online community	[13]
Twitter-2010	41,652,230	1,468,365,182	Twitter user profiles and social relations	[30]

Table 2: Graph datasets used in the experiments

The performance difference grows for larger graph instances as the numbers for the LiveJournal graph show where the Green-Marl program outperforms the other implementation by three to four orders of magnitude. This is due to the fact that the compilation time is a constant cost, while the aggregated cost of the API overhead grows with the size of the graph.

These results make it quite apparent that an approach based on using a compiled DSL program has superior performance over using an API due to the significantly lower overhead.

6.3 Impact of Compiler Optimizations

In Figure 7 we show the relative performance improvement that is achieved by applying the optimizations we presented in Section 5 on different algorithms. Table 1 contains the optimizations we used for the individual algorithms. We let them run for 120 minutes and extrapolated the total runtime if they did not finish in that timeframe to prevent them from running for days on larger graph instances. All experiments were executed using 36 threads except for Dijkstra, Fattest-Path and Tarjan which ran single-threaded since these algorithms are sequential in nature.

One can see that the effects on the performance depend on the optimization applied and on the algorithm itself. Closeness-Centrality and Betweenness-Centrality for example both benefit from MS-BFS but to a different extent. Closeness-Centrality only maintains a minimal context inside the BFS and can therefore use the maximum batch-size as it requires no extra memory. As a result it benefits the most from using MS-BFS. Betweenness-Centrality on the other hand requires a quite large context in the BFS which leads to an increased memory consumption. Consequently the system has to scale down the batch-size which decreases the benefit of MS-BFS. For example when running on the Twitter graph, the batch-size is reduced to 2 which has almost no benefit over using a regular BFS traversal.

Dijkstra and Fattest-Path have a very high speedup over the unoptimized version because the default `map` implementation used scales poorly for the `getKeyWithSmallestValue()` function which is crucial for these algorithms. Therefore the `priority-map` used in the optimized code not only results in faster code, but also in much better scaling regarding the graph size. This shows that the automatic selection of a better data-structure can save an unaware user from making the wrong choice which might not always be as obvious as in this case.

The moderate speedup for Tarjan and Kosaraju comes from replacing the `linked-list` by an `array-list`. The numbers are not as high as for Dijkstra due to the small difference in efficiency between the implementations and also the fact that the list is not as important for the performance in these two algorithms. This is another example how an automated optimization can provide better performance with no effort or interaction needed from the user.

The property merging performed for Pagerank and the Soman-and-Narang algorithm together with the precomputation of an inversed degree lead to a 1.3-2x speedup of the code due to fewer cache misses and a reduced contention in the floating point units.

Core operation in Adamic-Adar and Triangle-Counting is iterating over the common neighbors of two vertices. Using the improved common neighbor iteration as part of the optimization from Section 5.2 leads to an increased performance by 2.5-3.8x. Twitter shows a higher speedup than the other graphs because its data contains more skew which is handled better by the optimized version.

These results show the potential of graph specific optimizations when applied to Green-Marl procedures. Users benefit from these optimizations without having to implement them on their own as they are applied automatically and transparently. In addition, without sacrificing any performance, complexity of the code is reduced and potential sources of errors are removed.

7. FUTURE EXTENSIONS

In this section we describe our plans to extend PGX to tackle different challenges and to support different systems for graph analysis while keeping these details transparent from the user by using Green-Marl and PGQL.

7.1 Distributed Backend

Graph instances can grow to sizes where the data exceeds the capacity of the main memory of a single machine. We want to support such data sets and want to enable PGX to scale out in addition to scaling up. Therefore we have been developing a new backend for PGX [25] that allows distributed graph processing and analysis on a cluster of machines where the size of the graph can exceed the main memory of a single machine as long as it fits the combined memory. Implementing graph analytic algorithms for distributed systems is often hard due to their programming paradigms which have to accommodate the distributed model rather than offering an intuitive approach to graph algorithms [27]. The Green-Marl and PGQL DSLs on the other hand offer an intuitive way of implementing graph analysis and query algorithms. Previous work has shown that compiling imperative Green-Marl code for a distributed framework is possible and feasible without sacrificing performance [26].

Our plan is to extend the Green-Marl compiler to be able to generate code for the distributed backend to allow users to reuse their existing Green-Marl programs. This way users can run their graph analysis code both in shared-memory and a distributed environment without having to implement the same algorithm multiple times for different systems. Usually distributed systems such as GraphLab [31] support running programs on a single machine instead of a cluster, but this includes a significant amount of performance overhead since the programming paradigm was designed for a distributed system rather than a shared-memory environment [25]. Our system on the other hand allows using the same source code while having programs that are optimized for the specific system they are running on. Therefore the users benefit from an intuitive programming language for graph analysis algorithms while their programs are still portable and produce highly optimized code.

7.2 Database Backend

PGX stores the graph data in a relational database which enforces ACID properties for modifications on the graph, while graph analysis is performed in a main-memory runtime. This design allows high performance analysis but it comes with a certain overhead of loading and converting the graph data from the database into the in-memory representation. Previous work has shown that

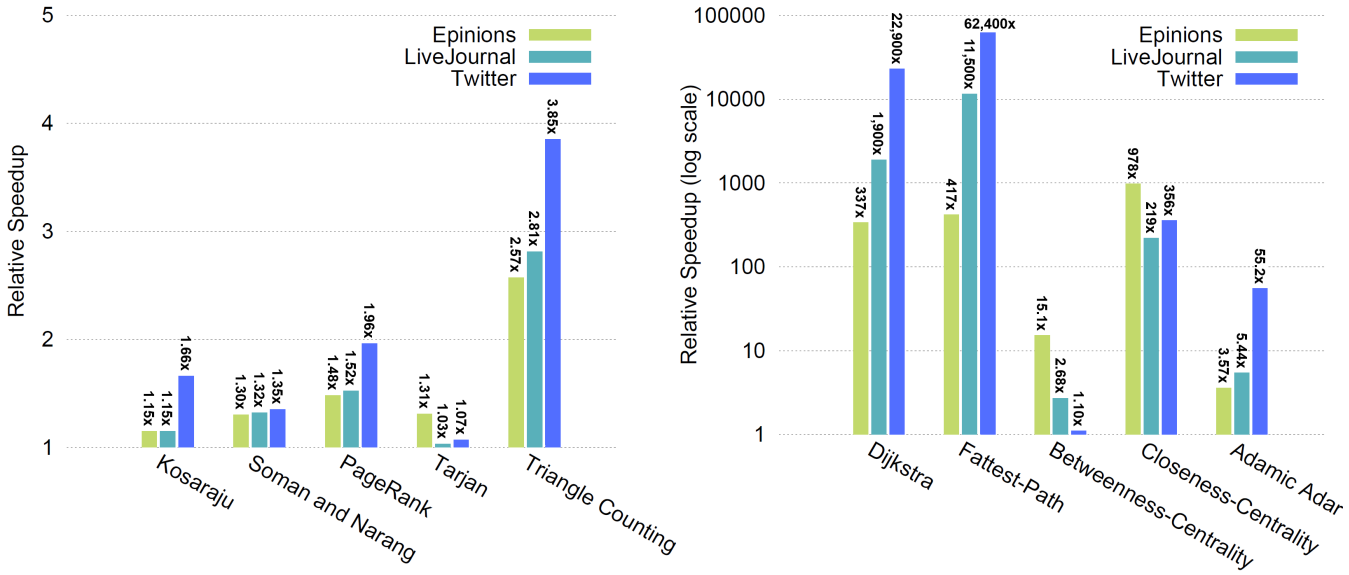

Figure 7: Relative speedup of optimized Green-Marl code using different data-sets.

graph analysis can be performed inside a database by implementing the algorithms in SQL or by using a different language such as Gremlin that is transformed into SQL statements [21, 40]. Performing graph analysis inside the database removes the cost of transferring the data to an external system while utilizing the highly optimized SQL query engine. This takes advantage of relational databases that have been around for decades and many have been highly optimized. Moreover, relational databases are well understood and are widely used in enterprise settings for both transactional and analytic workloads. Unfortunately the systems that implement this approach rely on the database optimizer alone to optimize their graph analysis programs, leaving out all optimizations that can be applied when using graph domain specific knowledge, such as the ones presented in this work. Hence, we plan to extend the Green-Marl compiler to be able to generate SQL code out of graph analysis programs. This will leverage graph specific optimizations as well as optimizations performed by the database optimizer that are specialized for the relational data model. We believe that this approach can help us achieve a performance superior to other systems that perform graph analysis inside a relational database.

8. CONCLUSION

In this paper we introduced our new graph database system which allows consistent management of graph data and fast in-memory analytics. Our system allows users to write their analysis algorithms and graph queries in an intuitive and flexible way and so improve their productivity. We use Green-Marl whose imperative model makes it an intuitive language for analytic algorithms and PGQL with its declarative model for queries and pattern matching. We showed how using a DSL enables the compiler to apply optimizations using the language's high-level semantics and that these optimizations can significantly improve the performance. Compiling and running DSL programs inside PGX avoids the runtime overhead of an API and we demonstrated how this can speed up graph analysis by several orders of magnitude compared to an implementation that uses the system API.

9. ACKNOWLEDGMENTS

We thank Felix Kaser, Jinha Kim, Korbinian Schmid and Alexander Weld (Oracle Labs) for their work and contributions to PGX, the PGQL and Green-Marl language and for reviewing this paper. We also thank Manuel Then (TU Munich) who - during his internship at Oracle Labs - implemented MS-BFS support in PGX and the Green-Marl compiler and who gave much valuable input to our project.

10. REFERENCES

[1] AllegroGraph. http://franz.com/agraph/allegrograph/.
[2] Apache Giraph Project. http://giraph.apache.org.
[3] Apache TinkerPop. http://tinkerpop.incubator.apache.org.
[4] Boost Graph Library (BGL). http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/index.html.
[5] Cypher - the Neo4j query Language. http://www.neo4j.org/learn/cypher.
[6] InfiniteGraph. http://www.objectivity.com/infinitegraph.
[7] Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html,2016.
[8] Java universal network/graph framework. http://jung.sourceforge.net.
[9] Neo4j graph database. http://www.neo4j.org/.
[10] NetworkX. https://networkx.github.io.
[11] Oracle Spatial and Graph, RDF Semantic Graph,. http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/rdfsemantic-graph-1902016.html.
[12] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.
[13] Stanford network analysis library. http://snap.stanford.edu/snap.
[14] Tinkerpop, Gremlin. https://github.com/tinkerpop/gremlin/wiki.

[15] Titan Distributed Graph Database. http://thinkaurelius.github.io/titan/.

[16] Virtuoso Universal Server. http://virtuoso.openlinksw.com/.

[17] Lada A. Adamic and Eytan Adar. Friends and neighbors on the web. *Social Networks*, 25(3):211–230, 2001.

[18] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems (TODS)*, 37(4):31, 2012.

[19] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[20] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. STINGER: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC)*, pages 1–5. IEEE, 2012.

[21] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. The case against specialized graph analytics engines. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[22] L. C. Freeman. Centrality in social networks: Conceptual clarification. *Social Networks*, 1(3):215–239, 1979.

[23] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*, pages 505–514. International World Wide Web Conferences Steering Committee, 2013.

[24] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *ASPLOS*, pages 349–362. ACM, 2012.

[25] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. Pgx.d: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 58:1–58:12, New York, NY, USA, 2015. ACM.

[26] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying scalable graph processing with a domain-specific language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 208:208–208:218, New York, NY, USA, 2014. ACM.

[27] Sungpack Hong, Jan Van Der Lugt, Adam Welc, Raghavan Raman, and Hassan Chafi. Early experiences in using a domain-specific language for large-scale graph analysis. In *First International Workshop on Graph Data Management Experiences and Systems*, page 5. ACM, 2013.

[28] Volker Kaibel and Matthias A. F. Peinhardt. On the bottleneck shortest path problem, technical report, ZIB-Report, 2006.

[29] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *IEEE International Conference on Data Mining (ICDM)*, pages 229–238, 2009.

[30] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600. ACM, 2010.

[31] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[32] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proceedings of the 2009 IEEE IPDPS*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[33] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD. Proceedings of the 2010 international conference on Management of data*, pages 135–146. ACM, 2010.

[34] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A performance evaluation of open source graph databases. In *Proceedings of the First Workshop on PPAA*, PPAA '14, pages 11–18, New York, NY, USA, 2014. ACM.

[35] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471. ACM, 2013.

[36] L. Page. Method for node ranking in a linked database, September 4 2001. US Patent 6,285,999.

[37] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[38] Martin Sevenich, Sungpack Hong, Adam Welc, and Hassan Chafi. Fast in-memory triangle listing for large real-world graphs. In *Proceedings of the 8th Workshop on Social Network Mining and Analysis*, SNAKDD'14, pages 2:1–2:9, New York, NY, USA, 2014. ACM.

[39] Jyothish Soman and Ankur Narang. Fast community detection algorithm with gpus and multicore architectures. In *IPDPS*, pages 568–579. IEEE, 2011.

[40] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1887–1901, New York, NY, USA, 2015. ACM.

[41] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, pages 607–614. ACM, 2011.

[42] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[43] M. Then, M. Kaufmann, F. Chirigati, T. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. The more the merrier: Efficient multi-source graph traversal. *Proc. VLDB Endow.*, 8(4):449–460, December 2014.

[44] Adam Welc, Raghavan Raman, Zhe Wu, Sungpack Hong, Hassan Chafi, and Jay Banerjee. Graph analysis: do we have to reinvent the wheel? In *First International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2013.

[45] M. Wittmann, T. Zeiser, G. Hager, and G. Wellein. Short note on costs of floating point operations on current x86-64 architectures: Denormals, overflow, underflow, and division by zero. *CoRR*, abs/1506.03997, 2015.